Report on the Language Knit: A Component Definition and Linking Language Version 1.0.0

Alastair Reid
School of Computing
University of Utah

http://www.cs.utah.edu/flux/

February 2001

Contents

Chapter 1

Report

1.1 Introduction

Knit is a component definition and linking language which can be used with little or no modification with C and assembly code. Knit supports component hierarchies, cyclic component dependencies, automatic scheduling of initializers and finalizers, an extensible constraint system to detect errors in component composition and cross-module inlining to largely eliminate the overheads of componentization.

This report is the specification of the Knit language and should be suitable for writing Knit programs and building implementations. It is *not* a tutorial on programming in Knit such as the *Knit User's Manual and Tutorial*, so familiarity with component programming concepts and the unit model is assumed. **You should read the Knit user's manual, especially the tutorial chapter, before reading the rest of this report.**

1.2 Lexical Structure

ToDo: *Explain how to define Knit environment variables such as SRCDIR?*

1.2.1 Notational Conventions

These notational conventions are used for presenting syntax:

[⟨*pattern*⟩]	optional
{ ⟨*pattern*⟩ }	zero or more repetitions
⟨*pattern*⟩+	one or more repetitions
⟨*pat1*⟩ | ⟨*pat2*⟩	choice

BNF-like syntax is used throughout, with productions having the form:

⟨*nonterm*⟩ ::= ⟨*alt1*⟩ | ⟨*alt2*⟩ | . . . ⟨*altn*⟩

Quoted identifiers and punctuation symbols are terminals (i.e., keywords), unquoted identifiers are nonterminals, all punctuation symbols with no special meaning are terminals.

Identifiers conform to the usual C rules. Knit is unusual in that (at the time of writing) it has many *keywords* but no *reserved words*. For example, the keyword '`imports`' has a special meaning when used at the start of a unit definition (Section 1.10) but it may also be used elsewhere as a normal identifier. This feature should be regarded as controversial and may change in the future.

1.2.2 Error Detection

When detecting errors, Knit strives to strike a balance between being so strict as to make the language unusable and being so lax as to accept and attempt to interpret meaningless or ambiguous definitions. Accordingly, we applied the following rules wherever possible in the design.

Sloppy Lists. All lists of items separated by punctuation (commas, semicolons, etc.) may have a trailing piece of punctuation. For example, one may write `[a, b, c,]` instead of `[a, b, c]`.

Lazy Ambiguity Resolution. Ambiguities are reported only if they affect the meaning of a unit not just if they exist. For example, it is acceptable to bind an identifier to two different entities if the bundle identifier is not actually referred to.

Order Insensitivity. In a list of items (declarations, bindings, etc.) where the order of the items does not directly affect the semantics, the order does not matter. For example, the order of declarations in a file is irrelevant but the order of arguments to a unit does matter.

1.3 Basic Semantic Concepts: Objects, Bundles, and Types

The overall goal of a Knit specification is to define a number of *object instances* and the interconnections between them. An *object* is either a function or a top-level variable. (Unlike conventional linking) there may be multiple copies of a particular object in a program: an *object instance* is one particular copy of an object.

Bundles serve to group a set of objects (or *bundle members*) together and to provide names for objects. An object can appear in multiple bundles, can appear several times in the same bundle (with different names each time) and can have a different name in each bundle it appears in.

Bundles are assigned types. A *bundletype* is just a list of the members of the bundle. Bundletypes form a subtype hierarchy based on inclusion: a bundletype bt_1 is a subtype of a bundletype bt_2 iff bt_1 is a subset of bt_2. (In the future, bundletypes may include the C type of a function or constraints (Section 1.8) and the notion of bundle subtyping will be changed accordingly.)

All units, whether atomic (Section 1.10.1) or compound (Section 1.10.2), import zero or more named bundles and export one or more named bundles.

1.4 Definitions, Files, and Directives

Unit specifications may refer to various other entities by name: bundletypes (Section 1.5); other units (Section 1.10); compilation flags (Section 1.6); properties (Section 1.8); and types (Section 1.8).

Each such entity occurs in its own namespace. For example, the identifier `example` could be used to refer to a bundletype, a unit, a set of compilation flags, a property and a type *in the same program*.[1]

⟨*defn*⟩ ::= ‘`bundletype`’ ⟨*bundleTIdent*⟩ ‘`=`’ ⟨*bundleType*⟩
| ‘`flags`’ ⟨*flagsIdent*⟩ ‘`=`’ ⟨*flags*⟩
| ‘`property`’ ⟨*propIdent*⟩
| ‘`type`’ ⟨*typeIdent*⟩ ‘`<=`’ ⟨*supers*⟩
| ‘`unit`’ ⟨*unitIdent*⟩ ‘`=`’ ⟨*unit*⟩

[1] In practice, Knit programmers find it helpful to add a `_T` suffix to bundletype identifiers although, strictly speaking, this is not necessary.

Definitions may be scattered across multiple files. Files may refer to other files using the ‘`include`’ directive. If an included file also contains include directives those will be processed unless the unit file has already been included.

All definitions are “visible to” definitions in any file which directly or indirectly includes the file or is included by the file.

⟨*file*⟩ ::= { ⟨*directive*⟩ | ⟨*decl*⟩ }
⟨*directive*⟩ ::= ‘`include`’ ⟨*fileName*⟩

The directory directive defines the *default directory* for atomic units (Section 1.10.1). Only one directory directive can be included per file and directory directives only apply to units occurring in that file.

⟨*directive*⟩ ::= ‘`directory`’ ⟨*dirName*⟩

Note that directory names may refer to Knit environment variables (Section 1.2) by writing `${name_of_variable}`. For example, it is common to start a file with a directive like this:

```
directory "${SRCDIR}"
```

and then define `SRCDIR` when invoking the Knit compiler.

1.5 Bundletypes

A bundletype is a set of bundle member names. Bundletypes can be defined as extensions of other bundletypes. This is equivalent to copying the definition of the extender into the extendee. Extension is often used to emphasize the relationship between bundletypes.

⟨*bundleType*⟩ ::= ⟨*bundleTIdent*⟩
| ‘{’ ⟨*btElement*⟩_1 ‘,’ ... ⟨*btElement*⟩_m [‘,’] ‘}’ $m \geq 0$
⟨*btElement*⟩ ::= ‘`extends`’ ⟨*bundleTIdent*⟩
| ⟨*memberIdent*⟩

Examples:

```
{ malloc, free }
{ extends Malloc_T, realloc }
```

1.6 Source Code

Objects are defined in source files which must be compiled to produce object files. Source code can consist of actual C or assembly language files or (usually short) sections of literal C code or (despite the terminological abuse) they can consist of object files.

Source files may specify a directory in which the files appear. If the directory is omitted, the default directory is used if one was specified in the same file.

⟨*sourceCode*⟩ ::= ⟨*fileSet*⟩ [‘`with`’ ‘`flags`’ ⟨*flags*⟩]
⟨*fileSet*⟩ ::= ‘`files`’ [⟨*dirName*⟩] ‘`{`’ ⟨*fileName*⟩_1 ‘`,`’ ... ⟨*fileName*⟩_m [‘`,`’] ‘`}`’ $m \geq 0$
| ⟨*literal C*⟩ [‘`with`’ ‘`flags`’ ⟨*flags*⟩]

Compilation generally requires “compilation flags” to direct the compiler. Compilation flags are strings which are passed to the compiler when compiling the source code.

⟨*flags*⟩ ::= ⟨*flagsIdent*⟩
| ‘`{`’ ⟨*flag*⟩_1 ‘`,`’ ... ⟨*flag*⟩_m [‘`,`’] ‘`}`’ $m \geq 0$
⟨*flag*⟩ ::= ‘`flags`’ ⟨*flagsIdent*⟩
| ⟨*string*⟩

Examples:

```
%{ const char* ip_address = "128.36.15.8"; %}

files{ "main.c", "eval.c" } with flags { "-DKNIT" }
```

1.7 Object Sets

Sets of objects can be constructed from bundles, by extracting objects from bundles or using set union (denoted by ‘`+`’) and set difference (denoted by ‘`-`’). Four special object sets are defined: ‘`imports`’ denotes the set of all imported objects, ‘`exports`’ denotes the set of all exported objects, ‘`inits`’ denotes the set of all initializers declared in this unit and ‘`finis`’ denotes the set of all finalizers declared in this unit.

⟨*objectSet*⟩ ::= ⟨*objectSetDiff*⟩_1 ‘`+`’ ... ⟨*objectSetDiff*⟩_m [‘`+`’] $m \geq 1$
| ⟨*atomicObjectSet*⟩ ‘`-`’ ⟨*atomicObjectSet*⟩
⟨*atomicObjectSet*⟩ ::= ⟨*bundleIdent*⟩
| ‘`{`’ ⟨*objectIdent*⟩_1 ‘`,`’ ... ⟨*objectIdent*⟩_m [‘`,`’] ‘`}`’ $m \geq 0$
| ‘`(`’ ⟨*objectSet*⟩ ‘`)`’
| ‘`imports`’
| ‘`exports`’
| ‘`inits`’
| ‘`finis`’

1.8 Constraints

Constraints are used to detect errors in component interconnections. Each object may have zero or more *properties* associated with it. Each property of an object can be assigned a type. Types are related by a subtyping relationship (a partial order) and unit definitions can contain constraints between properties which refer to that subtyping relationship. An error is reported if no solution can be found which both satisfies all the constraints assigns a unique minimal type to each property.

⟨*constraints*⟩ ::= {‘`constraints`’ ‘`{`’ ⟨*constraint*⟩_1 ‘`;`’ ... ⟨*constraint*⟩_m [‘`;`’] ‘`}`’} $m \geq 0$

⟨*constraint*⟩ ::= ⟨*property*⟩ '=' ⟨*property*⟩
| ⟨*property*⟩ '<=' ⟨*property*⟩
| ⟨*property*⟩ '>=' ⟨*property*⟩
⟨*property*⟩ ::= ⟨*typeIdent*⟩
| ⟨*propIdent*⟩ ⟨*atomicObjectSet*⟩

Examples:

```
constraints{ context exports <= NoContext };
constraints{ context exports <= context imports };
```

1.9 Initializers, Finalizers, and Dependencies

Initializers and finalizers are functions of type

```
int <function>(void)
```

that are to be run before (respectively, after) the first use of any of a set of objects. Initializers and finalizers indicate failure by returning a non-zero result.

Dependencies between symbols are used to propagate initializers and finalizers up the call chain.

⟨*initialization*⟩ ::= {'initializer' ⟨*objectIdent*⟩ 'for' ⟨*objectSet*⟩ ';'}
{'finalizer' ⟨*objectIdent*⟩ 'for' ⟨*objectSet*⟩ ';'}
{'depends' '{' ⟨*depend*⟩_1 ';' ... ⟨*depend*⟩_m [';'] '}' ';'} $m \geq 0$
⟨*depend*⟩ ::= ⟨*objectSet*⟩ 'needs' ⟨*objectSet*⟩
| ⟨*objectSet*⟩ '<' ⟨*objectSet*⟩

There are two forms of dependency:

a `needs` b means that all of b's initializers (respectively, finalizers) are initializers (respectively, finalizers) for a as well. That is, if a is needed, then b's initializers (respectively, finalizers) must be run and they must be run before (respectively, after) a. This is useful for expressing the idea that b will call a.

a `<` b means that if a's initializers (respectively, finalizers) are run, they must be run before (respectively, after) b. This is useful for expressing the idea that b will call a iff a is part of the system.

Knit schedules calls to the initializers and finalizers in an order that respects the dependencies. Scheduling is described in more detail in Section 1.14.

1.10 Units

Units define sets of objects (the *exports* of the unit) and their connection to other objects (the *imports* of the unit). Units are the building block for component reuse and so they contain all information one might need about the unit: constraints, initialization information, a hint as to whether or not the "flattening" optimization would be worthwhile and the definition of the objects themselves.

There are two kinds of unit: atomic units which contain source code; and compound units which instantiate and interconnect other units.

⟨*unit*⟩ ::= ‘{’
‘`imports`’ ‘[’ ⟨*import*⟩_1 ‘,’ ... ⟨*import*⟩_m [‘,’] ‘]’ ‘;’ $m \geq 0$
‘`exports`’ ‘[’ ⟨*export*⟩_1 ‘,’ ... ⟨*export*⟩_m [‘,’] ‘]’ ‘;’ $m \geq 1$
⟨*constraints*⟩
⟨*initialization*⟩
[(‘`flatten`’ | ‘`noflatten`’) ‘;’]
(⟨*atomicBody*⟩ | ⟨*compoundBody*⟩) [‘;’]
‘}’

1.10.1 Atomic Units

An *atomic unit* consists of a collection of C (or assembly) files with information about how to compile them and their externally visible properties.

By default, the names of bundle members in the imports and exports correspond directly to the names of objects in the source code. This default correspondence can be changed with a ⟨*rename*⟩ section which defines how bundle members are mapped to identifiers in the source code.

⟨*atomicBody*⟩ ::= ⟨*sourceCode*⟩_1 ‘;’ ... ⟨*sourceCode*⟩_m [‘;’] $m \geq 1$
{ ⟨*rename*⟩ }

⟨*rename*⟩ ::= ‘`rename`’ ‘{’ ⟨*renaming*⟩_1 ‘;’ ... ⟨*renaming*⟩_m [‘;’] ‘}’ ‘;’ $m \geq 0$

⟨*renaming*⟩ ::= ⟨*objectIdent*⟩ ‘`to`’ ⟨*CIdent*⟩
| ⟨*bundleIdent*⟩ ‘`with`’ ‘`prefix`’ ⟨*CIdent*⟩
| ⟨*bundleIdent*⟩ ‘`with`’ ‘`suffix`’ ⟨*CIdent*⟩

Example:

```
unit malloc_statistics = {
  import[ in  : Malloc_T, files : Files_T ];
  export[ out : Malloc_T ];
  ...
  rename{ in with prefix in_; out with prefix out_ };
}
```

As a guard against carelessness, atomic units are required to have a non-empty dependency list. On the rare occasion that no dependency is required, one might use this dependency:

```
{} needs {}
```

1.10.2 Compound Units

A *compound unit* consists of a set of *bundle bindings* which define new bundles in terms of imported bundles and unit instances. Some additional experimental features which are described in Section 1.12.

⟨*compoundBody*⟩ ::= ‘`link`’ ‘{’ ⟨*binding*⟩_1 ‘;’ ... ⟨*binding*⟩_m [‘;’] ‘}’ $m \geq 0$

Bindings may be given in any order and may be mutually recursive. Bindings may refer to bundle identifiers defined in the imports and exports refer to bundle identifiers defined in the bindings.[2]

1.10.3 Unit Instantiation

Unit instantiation creates a copy of a unit in which the imports of the unit are bound to a list of *argument bundles* and the exports of the unit are bound to a list of bundle identifiers.

There are two ways of specifying the argument bundles: with an ordered list of bundles which are matched *by position*; and with an unordered list of bundles labeled with bundle identifiers which are matched *by name* with the import of the same name in the unit being instantiated.

⟨*binding*⟩ ::= '[' ⟨*bundleIdent*⟩_1 ',' ... ⟨*bundleIdent*⟩_m [','] ']' '<-' ⟨*unitInstance*⟩ $m \geq 1$

⟨*unitInstance*⟩ ::= ⟨*unitIdent*⟩ '<-' (⟨*posArgs*⟩ | ⟨*nameArgs*⟩)

Matching Arguments by Position

⟨*posArgs*⟩ ::= '[' ⟨*posArg*⟩_1 ',' ... ⟨*posArg*⟩_m [','] ']' $m \geq 0$

⟨*posArg*⟩ ::= ⟨*arg*⟩

⟨*arg*⟩ ::= ⟨*bundle*⟩
| '(' ⟨*unitInstance*⟩ ')'

Example:

```
[printf] <- printf_unit <- [putchar,strcpy]
```

Matching Arguments by Name

⟨*nameArgs*⟩ ::= '{' ⟨*nameArg*⟩_1 ',' ... ⟨*nameArg*⟩_m [','] '}' $m \geq 0$

⟨*nameArg*⟩ ::= ⟨*bundleIdent*⟩ '=' ⟨*arg*⟩
| ⟨*bundleIdent*⟩

We say that a *pun* occurs if the name of the import argument in the instantiated unit matches the name of the actual argument. That is, if the argument is of the form $x = x$. Such puns may be abbreviated to just x.

Example:

```
[printf] <- printf_unit <- {putchar=console_putchar, strcpy}
```

[2] Exports may also refer to bundleIdentifiers defined in the imports but this feature is regarded as controversial and should usually be avoided.

Inline Units

Short and simple units can also be instantiated "inline": instead of declaring a unit at the top level, it can be written on the right hand side of a binding. Inline units are required to have no imports.

⟨*unitInstance*⟩ ::= 'unit' ⟨*unit*⟩

Example:

```
[abc] <- unit { imports[]; exports[ out : {a,b,c} ];
                depends{ exports + inits + finis  needs  imports };
                %{ int a=1, b=2, c=3; %}
              };
```

Anonymous Units

Exceptionally short and simple atomic units can be further abbreviated by simply writing some inline C code.

⟨*unitInstance*⟩ ::= ⟨*literal C*⟩

Since there is no list of imports or exports, the C code is required to consist of a list of variable declarations of the form:

```
<type> <identifier> = <value>;
```

Example:

```
[abc] <- %{ int a=1, b=2, c=3; %};
```

1.10.4 Constructing Bundles

Bundles can also be constructed from other bundles using a combination of reference to other bundle identifiers, selecting individual symbols from other bundles and bundle unions.

⟨*binding*⟩ ::= ⟨*bundleIdent*⟩ '=' ⟨*bundle*⟩

⟨*bundle*⟩ ::= ⟨*atomicBundle*⟩_1 '+' ... ⟨*atomicBundle*⟩_m ['+'] $m \geq 1$

⟨*atomicBundle*⟩ ::= ⟨*bundleIdent*⟩
| '{' ⟨*memberBinding*⟩_1 ',' ... ⟨*memberBinding*⟩_m [','] '}' $m \geq 1$
| '(' ⟨*bundle*⟩ ')'

⟨*memberBinding*⟩ ::= ⟨*memberIdent*⟩ '=' ⟨*objectIdent*⟩

1.11 Flattening

Knit supports an optimization known as "flattening" (or cross-module inlining) which can eliminate all or most of the function-call overhead between components. Flattening can be turned on or off for individual units using the '`flatten`' and '`noflatten`' annotations on the unit instance or in the unit definition. Unit instances "inherit" the flattening disposition of the unit that instantiates them with the "closest" annotation having precedence. That is, an annotation on a unit definition has precedence over an annotation on the unit instance which has precedence over the flattening disposition of the unit which instantiates it.

⟨*unit*⟩ ::= '`{`'
'`imports`' '`[`' ⟨*import*⟩_1 '`,`' ... ⟨*import*⟩_m ['`,`'] '`]`' '`;`' $m \geq 0$
'`exports`' '`[`' ⟨*export*⟩_1 '`,`' ... ⟨*export*⟩_m ['`,`'] '`]`' '`;`' $m \geq 1$
⟨*constraints*⟩
⟨*initialization*⟩
[('`flatten`' | '`noflatten`') '`;`']
(⟨*atomicBody*⟩ | ⟨*compoundBody*⟩) ['`;`']
'`}`'

⟨*unitInstance*⟩ ::= '`flatten`' ⟨*unitInstance*⟩
| '`noflatten`' ⟨*unitInstance*⟩

1.12 Experimental Features

The features described in this section are experimental but are considered important enough to be worth documenting in their relatively unfinished and unstable state. We expect that these features will change somewhat in the near future but that future versions of Knit will provide broadly equivalent functionality.

1.12.1 Globals

Declaring a bundle *global* in a compound unit implicitly adds that bundle to the imports of every unit instance (transitively) contained inside that unit.

This can be useful if all or most units in a system import a common set of bundles and you do not envisage wanting to change which bundles it imports. Examples include functions such as `memcpy` or a software floating-point implementation.

⟨*compoundBody*⟩ ::= {⟨*globals*⟩}
'`link`' '`{`' ⟨*binding*⟩_1 '`;`' ... ⟨*binding*⟩_m ['`;`'] '`}`' $m \geq 0$

⟨*globals*⟩ ::= '`global`' '`{`' ⟨*global*⟩_1 '`,`' ... ⟨*global*⟩_m ['`,`'] '`}`' '`;`' $m \geq 0$

⟨*global*⟩ ::= ⟨*bundleIdent*⟩

1.12.2 Defaults

Compound units can contain a list of *default bundles* to be used if a unit requires a bundle that the unit instance does not explicitly provide. This feature is only used when passing arguments by name and if the last argument to the unit is '`...`'. Default bundles are selected by bundletype. There must be a unique choice of default for each argument.

This can be useful if all or most units in a single compound unit import a common set of bundles or if many unit interconnections are "boring" and one wants to draw attention to a few "interesting" interconnections. It differs from *globals* in that defaults apply only to that unit and in that the definitions of units being instantiated must still explicitly list the bundles as imports.

⟨*compoundBody*⟩ ::= {⟨*globals*⟩}
{⟨*defaults*⟩}
'`link`' '`{`' ⟨*binding*⟩_1 '`;`' ... ⟨*binding*⟩_m ['`;`'] '`}`' $m \geq 0$

⟨*defaults*⟩ ::= '`default`' '`{`' ⟨*default*⟩_1 '`,`' ... ⟨*default*⟩_m ['`,`'] '`}`' '`;`' $m \geq 0$

⟨*default*⟩ ::= ⟨*bundleIdent*⟩

1.12.3 Packages and Glue

Compound units can contain a set of *glue units* to be instantiated. Glue units can be specified by listing individual unit identifiers or by listing *package identifiers* (which name sets of unit identifiers). The imports of these units are provided from the default bundles and the exports of the units are all provided as default units.

The form '`find`' ⟨*bundleType*⟩ can be used to refer to a default bundle with a given bundletype. This is useful if one wants to export the export of a glue unit.

⟨*compoundBody*⟩ ::= {⟨*globals*⟩}
{⟨*glues*⟩}
{⟨*defaults*⟩}
'`link`' '`{`' ⟨*binding*⟩_1 '`;`' ... ⟨*binding*⟩_m ['`;`'] '`}`' $m \geq 0$

⟨*glues*⟩ ::= '`glue`' '`{`' ⟨*glue*⟩_1 '`,`' ... ⟨*glue*⟩_m ['`,`'] '`}`' '`;`' $m \geq 0$

⟨*glue*⟩ ::= '`package`' ⟨*packageIdent*⟩
| ⟨*unitIdent*⟩

⟨*defn*⟩ ::= '`package`' ⟨*packageIdent*⟩ '`=`' ⟨*package*⟩

⟨*package*⟩ ::= '`{`' ⟨*glue*⟩_1 '`,`' ... ⟨*glue*⟩_m ['`,`'] '`}`'

⟨*atomicBundle*⟩ ::= '`find`' ⟨*bundleType*⟩

1.12.4 Deltas

Compound units can be defined as a modification to an existing unit by listing what parts of the unit are to be removed and which are to be added.

⟨*modUnit*⟩ ::= '`modify`' ⟨*unitIdent*⟩ '`deleting`' '`{`' ⟨*remunit*⟩ '`}`' '`adding`' '`{`' ⟨*addunit*⟩ '`}`'

⟨*remunit*⟩ ::= {'`imports`' '`[`' ⟨*bundleIdent*⟩_1 '`,`' ... ⟨*bundleIdent*⟩_m ['`,`'] '`]`' '`;`'} $m \geq 0$
{'`exports`' '`[`' ⟨*bundleIdent*⟩_1 '`,`' ... ⟨*bundleIdent*⟩_m ['`,`'] '`]`' '`;`'} $m \geq 0$
{'`constraints`' '`{`' ⟨*constraint*⟩_1 '`,`' ... ⟨*constraint*⟩_m ['`;`'] '`}`' '`;`'} $m \geq 0$
{'`depends`' '`{`' ⟨*depend*⟩_1 '`;`' ... ⟨*depend*⟩_m ['`;`'] '`}`' '`;`'}) $m \geq 0$
{⟨*globals*⟩}
{⟨*glue'*⟩}
{⟨*defaults*⟩}
{'`link`' '`{`' ⟨*bundleIdent*⟩_1 '`,`' ... ⟨*bundleIdent*⟩_m ['`,`'] '`}`' '`;`'} $m \geq 0$

⟨*addunit*⟩ ::= {‘`imports`’ ‘`[`’ ⟨*import*⟩_1 ‘`,`’ ... ⟨*import*⟩_m [‘`,`’] ‘`]`’ ‘`;`’} $m \geq 0$
{‘`exports`’ ‘`[`’ ⟨*export*⟩_1 ‘`,`’ ... ⟨*export*⟩_m [‘`,`’] ‘`]`’ ‘`;`’} $m \geq 0$
{‘`constraints`’ ‘`{`’ ⟨*constraint*⟩_1 ‘`;`’ ... ⟨*constraint*⟩_m [‘`;`’] ‘`)`’ ‘`;`’} $m \geq 0$
{‘`depends`’ ‘`{`’ ⟨*depend*⟩_1 ‘`;`’ ... ⟨*depend*⟩_m [‘`;`’] ‘`}`’ ‘`;`’} $m \geq 0$
{⟨*globals*⟩}
{⟨*glues*⟩}
{⟨*defaults*⟩}
{‘`link`’ ‘`{`’ ⟨*binding*⟩_1 ‘`;`’ ... ⟨*binding*⟩_m [‘`;`’] ‘`}`’ ‘`;`’} $m \geq 0$

Example:

```
unit Goodbye =
  modify Hello
  deleting {
    glue{ hello };
  } adding {
    glue{ goodbye };
  }
```

1.12.5 Documentation Comments

Documentation comments are a special form of comment that may be treated specially by a documentation generator. Since they are intended for documenting definitions, they are restricted to appearing at the top level.

⟨*defn*⟩ ::= ⟨*doc comment*⟩

Examples:

```
/*#
Bring down system on fatal error.

  void panic(const char *fmt, ...);

The panic() function terminates the running system.  The message fmt is a
printf(3) style format string.  The message is printed to the console.
#*/
bundletype panic_T = { panic }
```

1.13 Compilation

Compiling a unit (whether atomic or compound) generates an object file[3] such that:

[3]The current version of Knit actually generates a set of object file archives (i.e., ‘`.a`’ files) to allow a limited form of dead-code elimination. This may not be necessary with more modern versions of GNU `ld`.

- The symbols *imported* by the unit are either undefined or common variables in the object file.
- The symbols *exported* by the unit are either defined or common variables in the object file.
- The symbols `knit_init` and `knit_fini` are defined and the symbol `knit_progress` is undefined. These functions are used for initialization and finalization and are described further in Section 1.14.
- No other symbols are defined or undefined.[4]

1.14 Scheduling

The initializer, finalizer and dependency declarations of the unit (including all subunits) are used to determine the order in which initializers, finalizers and the exports of the unit can be run.

This information is used to partition the initializers and finalizers into a *schedule* which is a pair of lists is and fs such that:

- All initializers of exported symbols are in is.
- All finalizers of exported symbols are in fs.
- For each initializer or finalizer x, all initializers of x occur *before* x in is, fs and all finalizers of x occur *after* x in is, fs.

An error is reported if it is no such partition exists. This can happen if two initializers are initializers for each other.

Note that these rules *do not* imply that is only contains initializers or that fs only contains finalizers. For example, if the only use of the filesystem is in a finalizer, it is perfectly possible that the filesystem will not be initialized until after the main program runs.

The lists $is = [x_1, \ldots x_m]$ and $fs = [x_1, \ldots x_m]$ are used to generate the functions `knit_init` and `knit_fini` defined by:

```
int knit_<init or fini>(void)
{
  int rc;
  rc = x_1();
  knit_progress("x_1", rc);
  if (rc) return rc;
  ...
  rc = x_m();
  knit_progress("x_m", rc);
  if (rc) return rc;
}
```

The function `knit_progress` is used as a "progress indicator" during initialization and finalization: it is called after each initializer or finalizer returns. This function has type:

[4] Actually, many other symbols may be defined but they are prefixed with an identifier which makes it unlikely that they will conflict with object files not generated by Knit.

```
void knit_progress(const char* what, int rc);
```

ToDo: *Show standard use of initializers for Unix programs*

Appendix A

Syntax

These notational conventions are used for presenting syntax:

[⟨*pattern*⟩]	optional
{ ⟨*pattern*⟩ }	zero or more repetitions
⟨*pattern*⟩+	one or more repetitions
⟨*pat1*⟩ | ⟨*pat2*⟩	choice

BNF-like syntax is used throughout, with productions having the form:

⟨*nonterm*⟩ ::= ⟨*alt1*⟩ | ⟨*alt2*⟩ | ... ⟨*altn*⟩

Quoted identifiers and punctuation symbols are terminals (i.e., keywords), unquoted identifiers are non-terminals, all punctuation symbols with no special meaning are terminals.

Identifiers conform to the usual C rules. There are no "reserved identifiers": keywords can be used as normal identifiers anywhere that they have no special meaning.

All lists of patterns separated by commas (or semicolons) may have a trailing comma (or semicolon). For example, one may write `[a, b, c,]` instead of `[a, b, c]`. This optional punctuation is omitted from the grammar to avoid clutter.

A.1 Lexical Structure

ToDo: *The rules for comments and strings aren't quite right because 'any' ought to exclude the terminating characters*

⟨*file*⟩ ::= { ⟨*lexeme*⟩ | ⟨*whitespace*⟩ }

⟨*lexeme*⟩ ::= ⟨*ident*⟩ | ⟨*punctuation*⟩ | ⟨*string*⟩ | ⟨*integer*⟩ | ⟨*literal C*⟩ | ⟨*doc comment*⟩

⟨*whitespace*⟩ ::= (whitestuff)+

⟨*whitestuff*⟩ ::= ⟨*whitechar*⟩ | ⟨*comment*⟩ | ⟨*ncomment*⟩

⟨*whitechar*⟩ ::= ⟨*newline*⟩ | ⟨*formfeed*⟩ | ⟨*tab*⟩ | ⟨*space*⟩

⟨*comment*⟩ ::= '`//`' {⟨*any*⟩} ⟨*newline*⟩

⟨*ncomment*⟩ ::= '`/*`' {⟨*any*⟩} '`*/`'

⟨*doc comment*⟩ ::= '`/*#`' {⟨*any*⟩} '`#*/`'

⟨*literal C*⟩ ::= '`%{`' {⟨*any*⟩} '`%}`'

⟨*string*⟩ ::= '"' {⟨*any*⟩} '"'
⟨*ident*⟩ ::= (⟨*alpha*⟩ | '_') {⟨*idChar*⟩}
| ''' {⟨*any*⟩} '''
⟨*idChar*⟩ ::= ⟨*alpha*⟩ | ⟨*digit*⟩ | '_'
⟨*alpha*⟩ ::= 'a' | 'b' | ... 'z' | 'A' | 'B' | ... 'Z'
⟨*digit*⟩ ::= '0' | '1' | ... '9'
⟨*punctuation*⟩ ::= '.' | ',' | ':' | ';' | '(' | ')' | '[' | ']' | '{' | '}' | '=' | '<' | '<=' | '>=' | '<-' | '...'

A.2 Context Free Syntax

ToDo: *Hack the grammar package so that subscripts come out as subscripts*

⟨*file*⟩ ::= { ⟨*directive*⟩ | ⟨*decl*⟩ }
⟨*directive*⟩ ::= 'directory' ⟨*dirName*⟩
| 'include' ⟨*fileName*⟩
⟨*defn*⟩ ::= 'bundletype' ⟨*bundleTIdent*⟩ '=' ⟨*bundleType*⟩
| 'flags' ⟨*flagsIdent*⟩ '=' ⟨*flags*⟩
| 'unit' ⟨*unitIdent*⟩ '=' (⟨*unit*⟩ | ⟨*modUnit*⟩)
| 'property' ⟨*propIdent*⟩
| 'type' ⟨*typeIdent*⟩ '<=' ⟨*supertypes*⟩
| 'package' ⟨*packageIdent*⟩ '=' ⟨*package*⟩
| ⟨*doc comment*⟩
⟨*bundleType*⟩ ::= ⟨*bundleTIdent*⟩
| '{' ⟨*btElement*⟩_1 ',' ... ⟨*btElement*⟩_m [','] '}' $m \geq 0$
⟨*btElement*⟩ ::= 'extends' ⟨*bundleTIdent*⟩
| ⟨*memberIdent*⟩

⟨*sourceCode*⟩ ::= ⟨*fileSet*⟩ ['with' 'flags' ⟨*flags*⟩]
⟨*fileSet*⟩ ::= 'files' [⟨*dirName*⟩] '{' ⟨*fileName*⟩_1 ',' ... ⟨*fileName*⟩_m [','] '}' $m \geq 0$
| ⟨*literal C*⟩ ['with' 'flags' ⟨*flags*⟩]
⟨*flags*⟩ ::= ⟨*flagsIdent*⟩
| '{' ⟨*flag*⟩_1 ',' ... ⟨*flag*⟩_m [','] '}' $m \geq 0$
⟨*flag*⟩ ::= 'flags' ⟨*flagsIdent*⟩
| ⟨*string*⟩

⟨*constraints*⟩ ::= {'constraints' '{' ⟨*constraint*⟩_1 ';' ... ⟨*constraint*⟩_m [';'] '}'} $m \geq 0$
⟨*constraint*⟩ ::= ⟨*property*⟩ '=' ⟨*property*⟩
| ⟨*property*⟩ '<=' ⟨*property*⟩
| ⟨*property*⟩ '>=' ⟨*property*⟩
⟨*property*⟩ ::= ⟨*typeIdent*⟩
| ⟨*propIdent*⟩ ⟨*atomicObjectSet*⟩
⟨*supertypes*⟩ ::= ⟨*typeIdent*⟩
| '(' ⟨*typeIdent*⟩_1 ',' ... ⟨*typeIdent*⟩_m [','] ')' $m \geq 0$

⟨*initialization*⟩ ::= {‘`initializer`’ ⟨*objectIdent*⟩ ‘`for`’ ⟨*objectSet*⟩ ‘`;`’}
{‘`finalizer`’ ⟨*objectIdent*⟩ ‘`for`’ ⟨*objectSet*⟩ ‘`;`’}
{‘`depends`’ ‘`{`’ ⟨*depend*⟩ _1 ‘`;`’ ... ⟨*depend*⟩ _m [‘`;`’] ‘`}`’ ‘`;`’} $m \geq 0$

⟨*depend*⟩ ::= ⟨*objectSet*⟩ ‘`needs`’ ⟨*objectSet*⟩
| ⟨*objectSet*⟩ ‘`<`’ ⟨*objectSet*⟩

⟨*objectSet*⟩ ::= ⟨*objectSetDiff*⟩ _1 ‘`+`’ ... ⟨*objectSetDiff*⟩ _m [‘`+`’] $m \geq 1$
| ⟨*atomicObjectSet*⟩ ‘`-`’ ⟨*atomicObjectSet*⟩

⟨*atomicObjectSet*⟩ ::= ⟨*bundleIdent*⟩
| ‘`{`’ ⟨*objectIdent*⟩ _1 ‘`,`’ ... ⟨*objectIdent*⟩ _m [‘`,`’] ‘`}`’ $m \geq 0$
| ‘`(`’ ⟨*objectSet*⟩ ‘`)`’
| ‘`imports`’
| ‘`exports`’
| ‘`inits`’
| ‘`finis`’

⟨*unit*⟩ ::= ‘`{`’
‘`imports`’ ‘`[`’ ⟨*import*⟩ _1 ‘`,`’ ... ⟨*import*⟩ _m [‘`,`’] ‘`]`’ ‘`;`’ $m \geq 0$
‘`exports`’ ‘`[`’ ⟨*export*⟩ _1 ‘`,`’ ... ⟨*export*⟩ _m [‘`,`’] ‘`]`’ ‘`;`’ $m \geq 1$
⟨*constraints*⟩
⟨*initialization*⟩
[(‘`flatten`’ | ‘`noflatten`’) ‘`;`’]
(⟨*atomicBody*⟩ | ⟨*compoundBody*⟩) [‘`;`’]
‘`}`’

⟨*atomicBody*⟩ ::= ⟨*sourceCode*⟩ _1 ‘`;`’ ... ⟨*sourceCode*⟩ _m [‘`;`’] $m \geq 1$
{ ⟨*rename*⟩ }

⟨*rename*⟩ ::= ‘`rename`’ ‘`{`’ ⟨*renaming*⟩ _1 ‘`;`’ ... ⟨*renaming*⟩ _m [‘`;`’] ‘`}`’ ‘`;`’ $m \geq 0$

⟨*renaming*⟩ ::= ⟨*objectIdent*⟩ ‘`to`’ ⟨*CIdent*⟩
| ⟨*bundleIdent*⟩ ‘`with`’ ‘`prefix`’ ⟨*CIdent*⟩
| ⟨*bundleIdent*⟩ ‘`with`’ ‘`suffix`’ ⟨*CIdent*⟩

⟨*compoundBody*⟩ ::= {⟨*globals*⟩}
{⟨*glues*⟩}
{⟨*defaults*⟩}
‘`link`’ ‘`{`’ ⟨*binding*⟩ _1 ‘`;`’ ... ⟨*binding*⟩ _m [‘`;`’] ‘`}`’ $m \geq 0$

⟨*globals*⟩ ::= ‘`global`’ ‘`{`’ ⟨*global*⟩ _1 ‘`,`’ ... ⟨*global*⟩ _m [‘`,`’] ‘`}`’ ‘`;`’ $m \geq 0$

⟨*global*⟩ ::= ⟨*bundleIdent*⟩

⟨*glues*⟩ ::= ‘`glue`’ ‘`{`’ ⟨*glue*⟩ _1 ‘`,`’ ... ⟨*glue*⟩ _m [‘`,`’] ‘`}`’ ‘`;`’ $m \geq 0$

⟨*glue*⟩ ::= ‘`package`’ ⟨*packageIdent*⟩
| ⟨*unitIdent*⟩

⟨*package*⟩ ::= ‘`{`’ ⟨*glue*⟩ _1 ‘`,`’ ... ⟨*glue*⟩ _m [‘`,`’] ‘`}`’

⟨*defaults*⟩ ::= ‘`default`’ ‘`{`’ ⟨*default*⟩ _1 ‘`,`’ ... ⟨*default*⟩ _m [‘`,`’] ‘`}`’ ‘`;`’ $m \geq 0$

⟨*default*⟩ ::= ⟨*bundleIdent*⟩

⟨*import*⟩ ::= ⟨*bundleIdent*⟩ ‘`:`’ ⟨*bundleType*⟩

⟨*export*⟩ ::= ⟨*bundleIdent*⟩ ‘`:`’ ⟨*bundleType*⟩

⟨*binding*⟩ ::= ‘`[`’ ⟨*bundleIdent*⟩_1 ‘`,`’ ... ⟨*bundleIdent*⟩_m [‘`,`’] ‘`]`’ ‘`<-`’ ⟨*unitInstance*⟩ $m \geq 1$
| ⟨*bundleIdent*⟩ ‘`=`’ ⟨*bundle*⟩

⟨*unitInstance*⟩ ::= ⟨*unitIdent*⟩ ‘`<-`’ (⟨*posArgs*⟩ | ⟨*nameArgs*⟩)
| ‘`flatten`’ ⟨*unitInstance*⟩
| ‘`noflatten`’ ⟨*unitInstance*⟩
| ‘`unit`’ ⟨*unit*⟩
| ⟨*literal C*⟩

⟨*posArgs*⟩ ::= ‘`[`’ ⟨*posArg*⟩_1 ‘`,`’ ... ⟨*posArg*⟩_m [‘`,`’] ‘`]`’ $m \geq 0$

⟨*posArg*⟩ ::= ⟨*arg*⟩

⟨*nameArgs*⟩ ::= ‘`{`’ ⟨*nameArg*⟩_1 ‘`,`’ ... ⟨*nameArg*⟩_m [‘`,`’ ‘`...`’] [‘`,`’] ‘`}`’ $m \geq 0$

⟨*nameArg*⟩ ::= ⟨*bundleIdent*⟩ ‘`=`’ ⟨*arg*⟩
| ⟨*bundleIdent*⟩

⟨*arg*⟩ ::= ⟨*bundle*⟩
| ‘`(`’ ⟨*unitInstance*⟩ ‘`)`’

⟨*bundle*⟩ ::= ⟨*atomicBundle*⟩_1 ‘`+`’ ... ⟨*atomicBundle*⟩_m [‘`+`’] $m \geq 1$

⟨*atomicBundle*⟩ ::= ‘`find`’ ⟨*bundleType*⟩
| ‘`{`’ ⟨*memberBinding*⟩_1 ‘`,`’ ... ⟨*memberBinding*⟩_m [‘`,`’] ‘`}`’ $m \geq 1$
| ⟨*bundleIdent*⟩
| ‘`(`’ ⟨*bundle*⟩ ‘`)`’

⟨*memberBinding*⟩ ::= ⟨*memberIdent*⟩ ‘`=`’ ⟨*objectIdent*⟩

⟨*modUnit*⟩ ::= ‘`modify`’ ⟨*unitIdent*⟩ ‘`deleting`’ ‘`{`’ ⟨*remunit*⟩ ‘`}`’ ‘`adding`’ ‘`{`’ ⟨*addunit*⟩ ‘`}`’

⟨*remunit*⟩ ::= {‘`imports`’ ‘`[`’ ⟨*bundleIdent*⟩_1 ‘`,`’ ... ⟨*bundleIdent*⟩_m [‘`,`’] ‘`]`’ ‘`;`’} $m \geq 0$
{‘`exports`’ ‘`[`’ ⟨*bundleIdent*⟩_1 ‘`,`’ ... ⟨*bundleIdent*⟩_m [‘`,`’] ‘`]`’ ‘`;`’} $m \geq 0$
{‘`constraints`’ ‘`{`’ ⟨*constraint*⟩_1 ‘`,`’ ... ⟨*constraint*⟩_m [‘`;`’] ‘`}`’ ‘`;`’} $m \geq 0$
{‘`depends`’ ‘`{`’ ⟨*depend*⟩_1 ‘`;`’ ... ⟨*depend*⟩_m [‘`;`’] ‘`}`’ ‘`;`’}) $m \geq 0$
{⟨*globals*⟩}
{⟨*glue’*⟩}
{⟨*defaults*⟩}
{‘`link`’ ‘`{`’ ⟨*bundleIdent*⟩_1 ‘`,`’ ... ⟨*bundleIdent*⟩_m [‘`,`’] ‘`}`’ ‘`;`’} $m \geq 0$

⟨*addunit*⟩ ::= {‘`imports`’ ‘`[`’ ⟨*import*⟩_1 ‘`,`’ ... ⟨*import*⟩_m [‘`,`’] ‘`]`’ ‘`;`’} $m \geq 0$
{‘`exports`’ ‘`[`’ ⟨*export*⟩_1 ‘`,`’ ... ⟨*export*⟩_m [‘`,`’] ‘`]`’ ‘`;`’} $m \geq 0$
{‘`constraints`’ ‘`{`’ ⟨*constraint*⟩_1 ‘`;`’ ... ⟨*constraint*⟩_m [‘`;`’] ‘`)`’ ‘`;`’} $m \geq 0$
{‘`depends`’ ‘`{`’ ⟨*depend*⟩_1 ‘`;`’ ... ⟨*depend*⟩_m [‘`;`’] ‘`}`’ ‘`;`’} $m \geq 0$
{⟨*globals*⟩}
{⟨*glues*⟩}
{⟨*defaults*⟩}
{‘`link`’ ‘`{`’ ⟨*binding*⟩_1 ‘`;`’ ... ⟨*binding*⟩_m [‘`;`’] ‘`}`’ ‘`;`’} $m \geq 0$

⟨*fileName*⟩ ::= ⟨*string*⟩

⟨*dirName*⟩ ::= ⟨*string*⟩

⟨*objectIdent*⟩ ::= ⟨*objectIdent*⟩

⟨*CIdent*⟩ ::= ⟨*ident*⟩

⟨*bundleIdent*⟩ ::= ⟨*ident*⟩

⟨*memberIdent*⟩ ::= ⟨*ident*⟩
⟨*bundleTIdent*⟩ ::= ⟨*ident*⟩
⟨*unitIdent*⟩ ::= ⟨*ident*⟩
⟨*flagsIdent*⟩ ::= ⟨*ident*⟩
⟨*packageIdent*⟩ ::= ⟨*ident*⟩
⟨*propIdent*⟩ ::= ⟨*ident*⟩
⟨*typeIdent*⟩ ::= ⟨*ident*⟩
⟨*qualIdent*⟩ ::= [⟨*bundleIdent*⟩ ‘.’] ⟨*memberIdent*⟩